Submission for WIDM'02

ID#55

Honey, I Shrunk the XQuery!
— An XML Algebra Optimization Approach.

Xin Zhang, Bradford Pielech and Elke A. Rundesnteiner
Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
(xinz|winners|rundenst)@cs.wpi.edu

Abstract

Mapping of XML data into and out of relational database systems, including query processing over such virtual XML views that wrap relational sources, has become a topic of critical importance recently. The Rainbow XML data management system, being developed at WPI, focuses on the processing and optimization of XQuery queries against XML views of that relational data. For this, Rainbow's query model, XML Algebra Tree (XAT), has been designed. Because the XML formatting operators are interleaved with the computation operators, this XAT must then be optimized before being translated into SQL. For this, our computation pushdown technology splits the XAT into the XML-specific and SQL-doable operators with the later then being converted into SQL statements. However, the XAT after computation pushdown may contain unreferenced columns or unused operators. We show that these unneeded operators cannot be discovered by the relational engine after SQL generation. Leaving these operators in the tree would create unnecessarily large SQL statements that will slow down the overall execution.

Our main contributions to XML query processing, described in this paper, are threefold. One, we describe the XAT algebra for modeling XQuery expressions. Two, we propose rewriting rules to optimize XQueries by cancelling operators. Lastly, we describe a cutting algorithm that removes unreferenced columns and operators from the XATs. We have implemented the techniques discussed in this paper in the Rainbow system. Our experimental study compares the performance of execution before and after operator cancel out and before and after cutting.

Keywords: XML, query processing, algebra, XQuery, XML to relational, query optimization

1 Introduction

Motivation. XML [1] has emerged as a popular choice for exchanging structured data between web applications. Because of its balance of power and flexibility, it is quickly becoming the standard for data representation on the Web. Combining these strengths of the XML data model with the maturity of relational database technology in one larger system makes a lot of sense given the demand for both reliable, persistent storage as well as a flexible means to exchange data between applications.

Combining these technologies creates major challenges that must be tackled. One is to load, model, and extract XML data in and out of the relational backend system without losing the semantics of the data. Creating an XML view of the relational data has emerged as the leading architectual design to bridge the gap between these two different data models [2, 7, 5] because of the expressive power of XQuery to define such mapping views that wrap the relational source. Such views can limit the data exposed to an application and they also wrap the data in any XML format desired by the applications.

The second challenge is to efficiently process queries against these XML views. The end user can specify queries against such XML views using their favorite XML query language. The XML query engine would now need to rewrite the user query, combined with the mapping query, to efficiently compute what the user desires. The difficulty with this scenario arises due to the impedence mismatch between the language to query the XML views and the language to query the underlying relational database.

Using the same language to define and query the XML views allows us to merge the query that defines the view with the user query over that view to form one larger query. That larger query would need to be translated into SQL to be executed over the relational database. However, this query rewriting and translation at the syntax level is difficult to optimize [6]. Therefore, instead we propose an XML-based algebra tree (XAT) representation of queries as a basis for this query optimization and translation. Our XML query engine represents the XQuery as XAT algebra and then handles the merging of the user and view definition queries and the optimization of the query plan before SQL pushdown.

Running Example. The following XML, adapted from XML Use Cases [12], will be used as the running example throughout this paper. The XML document describes a price list of multiple books where each book has one title, one publication source, and one price are given in Figure 1. The example XQuery in Figure 2 computes the minimum price for each book in the price list, and returns the book's title with its minimum price. The query result is shown in Figure 3. Figure 4 depicts the view query built on top of the relational tables that virtually restores the original XML document view according to Figure 1, against which the user queries are specified. Please notice that in Figure 4, `<source>` and `<prices>` are not used by the user query depicted in Figure 2.

```
<prices>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.amazon.com</source>
    <price>65.95</price>
  </book>
  <book>
    <title>TCP/IP Illustrated</title>
    <source>www.bn.com</source>
    <price>69.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.amazon.com</source>
    <price>34.95</price>
  </book>
  <book>
    <title>Data on the Web</title>
    <source>www.bn.com</source>
    <price>39.95</price>
  </book>
</prices>
```

Figure 1: XML Data Example.

```
<result> { FOR $t IN
      distinct(document("prices.xml")/book/title)
  RETURN
    <booktitle>
      $t/text()
    </booktitle>
}</result>
```

Figure 2: User XQuery Example.

```
<results>
  <booktitle>TCP/IP Illustrated</booktitle>
  <booktitle>Data on the Web</booktitle>
</results>
```

Figure 3: Query Result of User XQuery.

```
<prices>
  FOR $book IN document("dxv.xml")/book/row,
      $store IN document("dxv.xml")/store/row,
      $prices IN document("dxv.xml")/prices/row
  WHERE $book/bid = $prices/bid AND
        $source/sid = $prices/sid
  RETURN
    <book>
      $book/title,
      $store/source,
      $prices/price
    </book>
</prices>
```

Figure 4: View XQuery to go from relational to XML

Problem Definition. This paper focuses on the challenges faced when trying to optimize the XML algebra tree (XAT) generated by merging the query tree for the view definition and the user query (depicted in Figures 4 and 2 in our running example respectively). The goal of this optimization step is rewrite the XAT into two portions such that the top of the query contains the XML-specific operators (such as those that create XML fragments) and the bottom half contains the SQL-doable operators (such as Select and Join). This separation ensures that the bulk of the computation is pushed down to the relational engine and only computation not typically doable by a standard relational engine is performed by the Rainbow engine.

However, just attempting to maximally push all SQL-like computation down does not necessarily correspond to an optimal execution plan. Once SQL is to be generated in the lower portion of the XML algebra tree (XAT), note that data may be extracted from the relational database that would not be required by the user's query. For example, the detailed information about authors and titles of the books they wrote in the view defined in Figure 4 are not of interest to the user as expressed by the user query in Figure 2.

Contributions. The core contributions of this paper are three fold. First, we design an XQuery algebra, called XAT, which is capable of capturing the core features of XQuery. Using this algebra, we build an XAT that represents a logical plan for execution of the XQuery. The second contribution are heuristic-based rewriting rules for the XAT algebra which define how two operators may be combined or swapped with one another. Lastly, we describe our two-part XAT rewriting strategy that optimizes the XAT logical query plan for efficient query execution either in main memory or against a relational database. First, the rewriter iteratively applies our rewriting rules in a bottom up fashion to remove intermediate XML fragment construction and to push computation down to the relational engine. Second, it applies our top-down cutting algorithm to identify and remove unused operators. This process will shrink the XAT, resulting in a quicker execution as supported by our experimental results, reported in this paper and our system demonstrated at ACM-SIGMOD 2002 [16].

Outline. In the next section we briefly review the data model and operators of our XAT algebra. Section 3 introduces the Rainbow framework, in particular we review XAT generation, decorrelation and query merging. Query rewriting rules are given in Section 4. Section 5 describes the redundant operator cutting algorithm. Section 6 reports the experimental study. Related work and conclusions are given in Sections 7 and 8 respectively.

2 XML Algebra Tree (XAT)

2.1 XAT Data Model Components

The operators in the XAT must pass intermediate results to one another, which are ordered collections of related XML fragments. For this reason, the XAT data model is a tuple based ordered table called XAT table. Inspired by W3C's XQuery 1.0 Formal Semantics [13], every cell of a tuple is one of the following types:

- An atomic value
- A node including XML Element, XML Document, and XML Attribute
- An unordered or ordered collection of items (i.e, any mixture of nodes and atomic values).

Every column is denoted by a column name, which can be either a variable binding from the user-specific XQuery, e.g., $\$v$, or otherwise an internally generated variable, e.g., col_n. Every column is typed by an XQuery type defined by W3C [13].

2.2 Overview of XAT Operators

The XAT algebra has two purposes. First, it is used to explicitly represent the semantics of XQuery by composing well-defined building blocks, namely algebra operators. Second, it is used as foundation for the query engine in the Rainbow [4] system to allow for query optimization and processing. Every XAT operator's input and output are XAT tables defined in Section 2.1.

The XAT operators are classified into three categories: XML operators, SQL operators, and special operators. XML operators are used to represent the XML document related operations, e.g., navigation and construction. They are expose, tagger, navigate, aggregate, and XML set operators. SQL operators correspond to the relational complete portion of the algebra. They are project, select, Cartesian product, joins (theta, outer), distinct, groupby, orderby, and set operators. Special operators include operators used temporarily in different phases of optimization and operators shared by the class of XML and of SQL operators. They are SQLstmt, function, source, name, FOR, if-then-else, and merge operators. A brief description of each operator is given in Appendix A, while a detailed discussion can be found in [14].

3 Framework of Rainbow

3.1 Loading and Extracting of XML Data

The XML management system, Rainbow, has been designed to exploit relational database technology to manage XML data based on a flexible mapping strategy [15]. The Rainbow system is composed of three subsystems, e.g., a loading manager, a mapping manager, and an XML query engine build on top of a relational database. Rainbow supports flexible mapping between XML data and relational data in the loading manager. The system can flexibly switch between several different mapping strategies by changing the XQuery loading expression. Then, the mapping manager provides an extraction view query for each loading query. This view query is capable of reconstructing the original XML document.

3.2 Query Engine

The *Rainbow Query Engine*, the core part of the *Rainbow* system, processes XML queries. Rainbow's query engine uses the XML algebra, introduced in Section 2.2, for optimization and execution of queries. Below we briefly describe the XQuery to SQL translation mechanism, while Sections 4 and 5 describe the rewrite engine and cutting algorithm, respectively.

Figure 5 depicts the *Rainbow Query Engine* that uses the *Kweelt Parser* [11] to parse XQuery expressions. The *XAT Generator* analyzes the parsed tree and generates an *XML Algebra Tree* (XAT) for both the user and mapping queries. The generated XATs are then unnested by the *XAT Decorrelator*. The two XATs are merged by the *XAT Merger*, generating one final merged XAT that has the user's query on top of the mapping query. See Figure 6 for the XAT at this point. The *XAT Rewriter* rewrites the merged XAT so as to push as much translatable into SQL computation down to the bottom of the tree (see Figure 7). *SQL Generator* generates SQL from the bottom portions of XAT, which is then executed against the underlying relational database by the *XAT Executor*. The tuples returned by the SQL engine are then tagged into XML elements

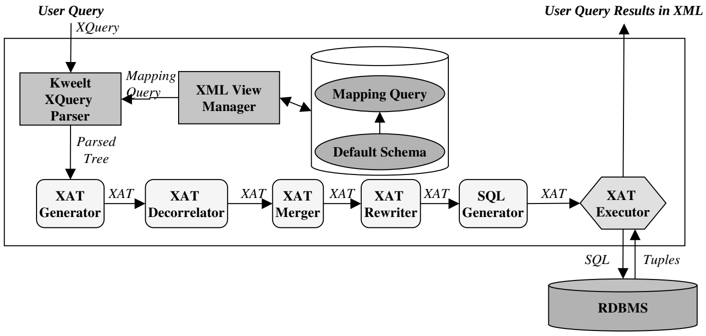

Figure 5: Architechure of Rainbow Query Engine.

using construction information from XML specific operators at the top in XAT. Finally, the results in the form of an XML document are returned back to the user.

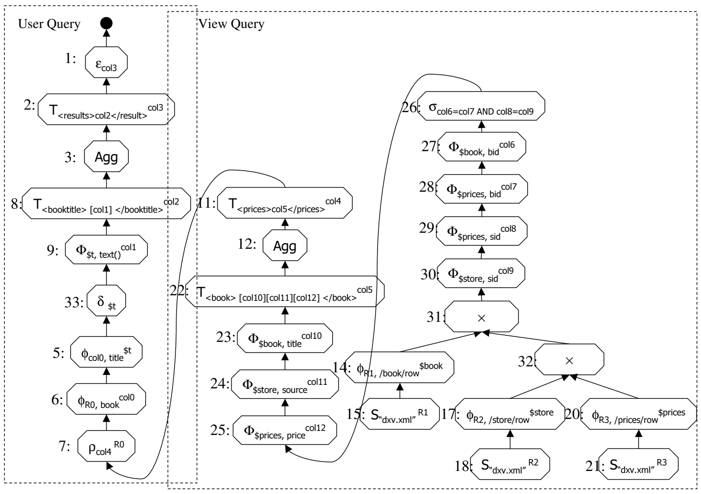

Figure 6: XAT after Generation, Decorrelation, and Merging

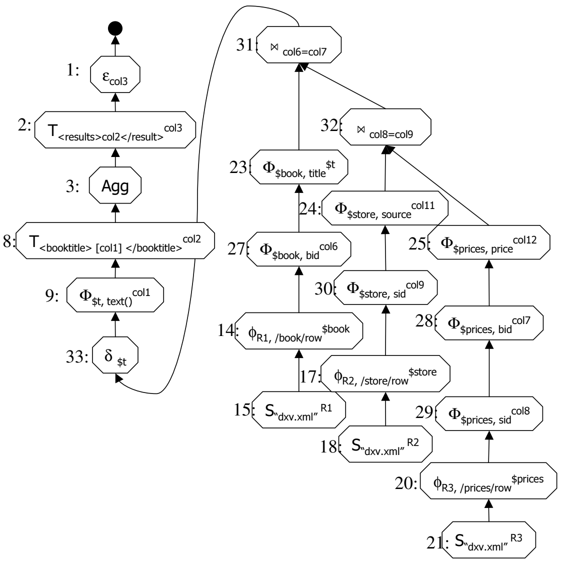

Figure 7: XAT after Rewriting.

4 Query ReWriting

Query rewriting in Rainbow refers to the process of evaluating equivalence rules on an XML algebra tree (XAT) to optimize the tree for efficiency and create a version of the tree where SQL queries can easily be generated. The administrator can choose from several possible rule application heuristics in order to optimize the tree, otherwise default heuristics are chosen.

4.1 XAT Traversing

The XAT Traverser uses a bottom up approach to find an operator to rewrite. The bottom up approach is necessary because operators are moved downwards in the tree. If top down were used, the Traverser would try to push down operators before their dependencies have been rewritten. The operators would be stopped prematurely from being moved maximally down the tree. The Traverser iterates over the tree and applies equivalence rules according to one of several heuristics (Section 4.3). During rewriting, the engine only examines the local operator it is pushing down (x1) and its child (x2). The appropriate rule is chosen based on the types of x1 and x2 and the heuristic. After evaluating the rule, x1 and x2 are typically swapped in the tree, this has the affect of pushing x1 further down in the tree.

4.2 Common ReWrite Rules

This section details some of the most commonly used rewrite rules in Rainbow. These rules state that Navigate operators can be pushed through all operators until a dependency on its child operator arises:

```
φ(x1, path):x2[φ(y1, path):y2[T]] → φ(y1, path):y2[φ(x1, path):x2[T]] (x1 != y2)
φ(x1, path):x2[σ(c)[T]]→ σ(c)[φ(x1, path):x2[T]]
φ(x1, path):x2[×[T1, T2]] → ×[T1, φ(x1, path):x2[t2]] (if x1 in DM(T2))
φ(x1, path):x2[×[T1, T2]] → ×[φ(x1, path):x2[T1], T2] (if x1 in DM(T1))
```

Furthermore, during the push down process, redundant operators may be canceled out. For example, the Navigate operator can be canceled with the Tagger operator, but not vice versa.

```
φ(x, /):y[T(<tag>[z]</tag>):x[s]]→ s
```

4.3 ReWrite Heuristics

Several heuristics can be chosen to rewrite XAT into the normalized form where all XML operators are on the top and all SQL-doable are at the bottom. The order that the heuristics are applied can affect the resulting tree.

1. **Removal of the construction of intermediate XML fragments:** The mapping query generates bags of XML data using the Navigate operators and then XML fragments are formed later from those bags by the Tagger operators. The user query then navigates into said constructed fragments and builds additional bags with Navigate operators and then more bags containing XML fragments with more Taggers operators. This process is redundant as the same information is stored multiple times in intermediate result sets. This redundancy can be eliminated by combining the different Navigate and Tagger operators from the user and mapping query. Once the Navigate pairs have been identified, we can rename the mapping query Navigates to what is specified in the user query and then remove the user query

Navigates.

This heuristic is best illustrated by an example. In Figure 6, operator #23 navigates from $book to title and binds that output bag to col10. Then, operator #21 creates a book element out of col10, col11 and col12 (all 3 columns are elements). Using the equivalence rules for operator #23 and its child listed above, operator #23 is pushed down. Figure 7 shows the final position of operator #23 in the XAT. We then iterate over the tree looking for another Navigate to push down. After rewriting operator #6, the Traverser next chooses operator #5. This operator #5 navigates from col0 (which represents R0/book) to title and binds the result to $t. Operator #5 is equivalent to operator #23. The rewriter determines this by evaluating equivalence rules for #5 and its children, including for #5 and #22. The rule says that the Navigate #5 unnests the title element because its destination is title. The rule then deletes the title element from the Tagger operator and then deletes #5 and replaces it with a NameColumn operator. This NameColum operator #34, not shown in the final tree due to space renames col10 into $t. The traverser then searches for another Navigate operator to push down until none can be found.

2. **Cancel Duplicate Navigate Operators:** An XAT may contain one or more equivalent Navigates. These duplicates may exist in both the user and the mapping query portion of the tree and are not necessarily removed in Heuristic 1 because they do not relate to the construction of intermediate XML fragments. The ReWrite engine identifies these operators by comparing the Navigates. If the entry point and destination match, then the top most Navigate is replaced with a NameColumn operator. This NameColumn will rename the output column from the lower Navigate into that of the upper. Figure 8 shows an example of a query with duplicate Navigates. Noticed the $t/text() in the WHERE clause and in the RETURN clause. These will each create a Navigate operator with $t/text() as the path.

```
<result>{
  FOR $t IN distinct(document("prices.xml")/book/title)
  WHERE $t/text() = "Data on the Web"
  RETURN <booktitle>$t/text()</booktitle>
}</result>
```

Figure 8: XQuery Example with Duplicated Navigations.

3. **Computation Pushdown:** Relational engine SQL operators such as Distinct, Select, GroupBy, Join, and Union can be pushed down from the user portion of the query tree into the mapping portion by repeatedly evaluating equivalence rules. In the running example, the SQL computation that is pushed down is the Distinct operator (operator #33).

4. **Propagate Renames:** Some of the above heuristics will create NameColumn operators in the tree. These operators are elminated by pushing them down in the tree and changing the bindings of the corresponding Navigate and Select operators. In the running example, after Heuristic 1 is applied,

there are several NameColumn operators in the tree including operator #34 (see above). The traverser starts from the bottom of the tree and moves up until it finds the first NameColumn to push down. Eventually, it finds operator #34 and then rules are evaluated until the NameColumn reaches the bottom of the tree. Anytime the child of operator #34 contains an instance of the old column name (col10) in the operators path or data model, that instance is renamed into the new binding, $t. Notice the binding change in operator #23 in Figure 7. The NameColumn operators are deleted when they reach the bottom of the tree.

5 Schema Cleanup and Unused Operator Cutting

5.1 Schema Cleanup

XAT Schema cleanup is used to optimize the size of the intermediate result generated by each operator, reducing the overall space used during XAT execution. It will get rid of the unused columns in the XAT tables. This can also get rid of redundant operators. Every operator consumes some and produces other columns. Therefore, by recursively analyzing each operator in the XAT, we can compute the columns required for each operator. Table 1 describes the *consumed*, *modified*, and *produced* columns for the most commonly used operators. For the full description, see [14].

Operator	Consumed	Produced	Modified
$T_p^{col}(S)$	Columns used in pattern p	col	N/A
$\phi_{col,path}^{col'}(S)$	col	col'	All
$\Phi_{col,path}^{col'}(S)$	col	col'	N/A
$Agg(S)$	N/A	N/A	All
D_{col}	col	N/A	All
$E_{desc}^{col[1..n]}$	col[1..n]	$col[1..n]$	N/A
$S_{desc}^{col[1..n]}$	N/A	$col[1..n]$	N/A
$\pi_{col[1..n]}(S)$	$col[1..n]$	N/A	N/A
$\sigma_c(S)$	columns in condition c	N/A	All
$\bowtie_c(ls, rs)$	columns in condition c	N/A	All
$\gamma_{col[1..n]}(S, sq)$	$col[1..n]$ and columns in sq	columns generated in sq	?

Table 1: Consumed, produced, and modified columns for each operator type.

The minimum schema for each operator, MS, contains the columns that the operator consumes, plus those consumed by the operators above it. Hence, we have:

$MS := (MS_p - P_p) \cup C_p \cup (P - C)$

where P and C represent the columns produced and consumed by the operator whose MS we wish to calculate. MS_p is the minimal schema for the operator's parent. P_p and C_p are the set of columns produced and consumed, respectively for the operator's parent. Table 2 shows the schema for the example in Figure 7, using the formula depicted above. During execution, each operator uses this table to determine how its output schema should look like. This avoids adding project operators into the XAT.

Node	Produce	Consume (C_l, C_r)	Schema Before Cutting	Schema After Cutting (l, r)
1	{}	{col3}	{$col3}	{col3}
2	{col3}	{col2}	{R1, $book, col6, R2, $store, col9, R3, $prices, col8, col7, col12, col11, $t, col1, col2, col3}	{col3}
...	...	...	...	...
31	{}	({col6}, {col7})	{R1, $book, col6, $t, R2, $store, col9, col11, R3, $prices, col8, col7, $col12}	({$t}, {})
...	...	...	...	...
27	{col6}	{$book}	{R1, $book}	{$book, $col6}
15	{R1}	{}	{R1}	{R1}
32	{}	({col9}, {col8})	{R2, $store, col9, col11, R3, $prices, col8, col7, col12}	({}, {$col7})
25	{col12}	{$prices}	{R3, $prices, col8, col7, col12}	{col8, col7, col12}
...	...	...	...	...
18	{R2}	{}	{R2}	{R2}
...	...	...	...	...
21	{R3}	{}	{R3}	{R3}

Table 2: Schema Cleanup for the XAT in Figure 7.

5.2 Unused Operator Cutting

After the above schema cleanup of the XAT, some operators may produce data that will not be consumed (used) by the operators above it in the XAT. Therefore, those operators are not necessary for the final query result. We can cut those operators to improve the query execution performance.

Our technology is called **cutting matrix**. Our *cutting matrix* is composed of operator identifications as row header and column names as column header. There are six symbols, i.e., *r, c, p, m, x, _*, used in the matrix, where *r* means required column, *c* means consumed column, *p* means produced column, *m* means modified column, _ means possible affected column, and *x* means cutable operator. For example, the cell at location $(1, r1)$ has a r, means $r1$ is a required column for operator $\#1$.

We first fill the *cutting matrix* with *p, c, _* to show column names which each operator produced, consumed and possibly affected. Then for each operator, if a column produced lower in the tree used higher in the tree, and that column is possible to be affected by the current operator, then, we put a *m* there to show it is modified. For example, the cell $(23, col6)$ in Figure 9 means that the operator #23 modified the $col6$, which is produced by operator $\#27$ and consumed by operator $\#31$.

We then update the first row of the matrix with required columns used to generate the final result, which is the cell $(1, col3)$ in Figure 9. The rules to compute the r is as follows:

```
(A,Y)=r
(A,X)=r IF (B,X)=c and ((B,Y)=p or (B,Y)=m), (A,Y)=r
```

They mean if the required column is modified or produced by another operator, then all other consumed columns of that operator are required. In our case, we set cell $(1, col3) = r$ as the initial condition. Then,

	r1	$book	col6	$title	r2	$store	col9	c11	r3	$prices	col8	col7	col12	col1	col2	col3	cut?
1	r	r		r										r	r	r	
2															c	p	
3	_	_	_	_	_	_	_	_	_	_	_	_	_	_	m	_	
8														c	p		
9	_	_	_	c	_	_	_	_	_	_	_	_	_	p	_	_	
33	_	_	_	c	_	_	_	_	_	_	_	_	_	_	_	_	
31	_	_	c		_	_	_	_	_	_	_	c	_	_	_	_	x
23	_	c	m	p	_	_	_	_	_	_	_	_	_	_	_	_	
27	_	c	p	_	_	_	_	_	_	_	_	_	_	_	_	_	
14	c	p															
15	p																
32	_	_	_	_	_	_	c	_	_	_	c	m	_	_	_	_	x
24	_	_	_	_	_	c	m	p	_	_	_	m	_	_	_	_	x
30	_	_	_	_	_	c	p	_	_	_	_	m	_	_	_	_	x
17					c	p											x
18					p												x
25	_	_	_	_	_	_	_	_	_	c	m	m	p	_	_	_	x
28	_	_	_	_	_	_	_	_	_	c	m	p	_	_	_	_	x
29	_	_	_	_	_	_	_	_	_	c	p	_	_	_	_	_	x
20									c	p							x
21									p								x

Figure 9: Operator Cutting Analysis Tables.

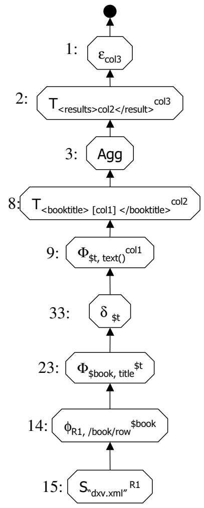

Figure 10: Cutted XAT in Figure 7

the columns $col2$, $col1$, $title$, $book$ are all *required*.

The last step is computing the $cut?$ column in the matrix, which is by the rule:

```
A is cutable IF for any X in ((A, X)=p or (A, X)=m) there is no (1,X)=r
```

It means none of the produced or modified columns by that operator are used in producing the final result. We compute cutting matrix in Figure 9 for the XAT depicted in Figure 7. As we can see that any operators, the operators #31 #32 #24 #30 #17 #18 #25 #28 #20 #21, deemed unnecessary by our formulas can safely be removed from the tree leaving behind a still correct and equivalent tree, as shown in Figure 10.

6 Experiments

We have implemented the Rainbow system in Java JDK1.2 using Xerces for DOM Parser, and Kweelt Engine [11] to generate the parsed tree for XQuery. The rewriting rules are written in Java classes and can be loaded dynamically, depending on the situation.

Experiment Setup. We use the XQuery statements described in Kweelt's test cases, which in turn were derived from the W3C's test cases. We also designed synthetic data sets with different sizes. All experiments are run on a PIII800 machine with 256 MB memory under Windows 2000 Professional.

Performance Gain. In the first experiment, we compare four kinds of query execution performances. In Figure 11, the x-axis shows increasing data sizes in terms of the number of elements, while the y-axis shows the total execution time for the queries. We use m to denote the number of operators and n the data size.

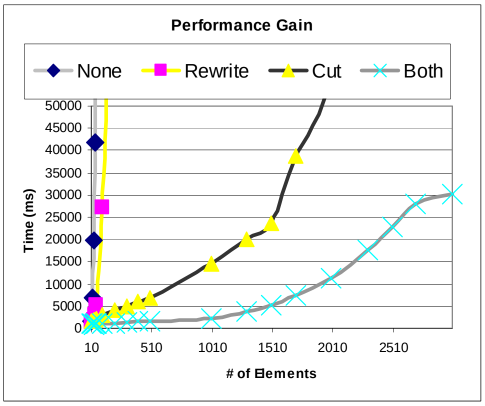

Figure 11: Performance Comparison

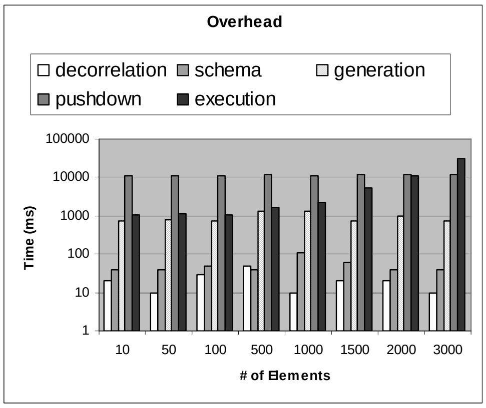

Figure 12: Overhead of Schema Cleanup

As we can see, the original performance without any optimization reaches the memory limits really fast $(O(m * n^2))$ and then the performance decreased dramatically because of the memory swapping. Rewriting only can reduce the number of nodes in the XAT somewhat when such redundant operators happen to exist. Therefore the XAT can handle more data within the same amount of memory. However, there may still be redundant columns. Hence, it's also $(O(m * n^2))$. The performance with schema clean up only scales much better, which is roughly $(O(m * n))$. As expected, as the performance with both the schema clean up and cutting is the best.

Optimizer Overhead. In this experiment we study the overhead of the schema cleanup and the rewriting. In Figure 12, the X-axis denotes the data size, the Y-axis the time (ms) in log scale. We can see the overhead of the operator cancel out and the schema clean up are constant for same query and relatively small compare to the execution time. Hence, they can be neglected for large data sets.

7 Related Work

Commercial relational systems including Oracle, SQL Server, and DB2 have or are adding support for XML data. Oracle's XMLSQL utility [10] is capable of storing XML data, using a fixed mapping, in its object-relational engine and then queries may be issued against these objects using XSQL, an extention of SQL that supports XPath expressions. Oracle relies on an additional XSLT script to generate the return XML while this element construction is explicit in XQuery. Microsoft's SQL Server 2000 [8] is capable of evaluting XPath expressions over XML views, but not joins or nested queries. IBM's DB2 Data Access Definition (DAD) [3] language generates an XML document using arbitrary selections and joins on existing relational tables. However, the language lacks support for user-defined functions and nested queries.

Similar to Rainbow, both XPERANTO [2] and SilkRoute [5] also process XQueries over virtual XML views of relational data using a library of rewrite rules that push computation down to the relational engine. However, key differences include the data model for XPERANTO, which is the XML Query Graph Model XQGM that does not support collections and is not order-sensitive. XQGM models element construction and navigation using different functions in the project operator. XQGM uses function-composition for cancel out, whereas we propose rule-based optimization for the XAT. Lastly, Rainbow proposes algorithms for redundant join operator cutting while XPERANTO provides only the traditional predicate cutting. Silkroute [5] focuses on generating efficient SQL queries that exploit one-to-one functional dependencies to reduce the number of sorted outer-union SQL queries. In contrast, Rainbow focuses on optimization, such as cutting, at the XML algebra-level before the generation of SQL queries.

AGORA [7] translates XQuery queries over XML views into equivalenct SQL queries by rewriting at the XQuery syntax level, not using an algebra. Niagara [9] proposes an XML algebra for the purpose of efficiently processing queries over XML data, not over relational systems. Niagara also proposes a set of equivalence rules, some of which were adopted in the creation of Rainbow's rule library.

8 Conclusions

This paper explains Rainbow's approach to efficient processing of XML queries over virtual XML views of relational data. We describ our XML algebra that captures the semantics of XQuery. Using this algebra, we propos a 2-fold optimization process at the logical XML algebra level. First, equivalence rules are applied according to an array of heuristics in order to push as much computation down to the SQL. Second an algorithm for cleaning the schema for each operator and the cutting of unused operators has been designed. These two techniques together result in a more efficient XAT that can then be converted into compact SQL queries. Preliminary experimental results confirm that the query execution time was significantly quicker after optimization.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensaible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml, Feburary 1998.

[2] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *VLDB*, pages 646–648, 2000.

[3] DB2 UDB XML Extender. XML Extender Administration and Programming. http://www-4.ibm.com/software/data/db2/extenders/xmlext/library.html, December 1999.

[4] DSRG. Rainbow: Relational Database Auto-Tuning for Efficient XML Query Processing. http://davis.wpi.edu/dsrg/rainbow, 2002.

[5] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[6] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[7] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with xml and relational. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[8] Microsoft Corp. Microsoft sql server. http://www.microsoft.com.

[9] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, F. Tian, J. Shanmugasundaram, C. Zhang, R. Ramamurthy, B. Jackson, Y. Wang, A. Gupta, and R. Chen. The Niagara Internet Query System. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'00)*, 2000.

[10] Oracle Technologies Network. Using XML in Oracle Database Applications. http://technet.oracle.com/tech/xml/info/htdocs/otnwp/about_oracle_xml_products.htm, November 1999.

[11] A. Sahuguet. Kweelt: More than just "yet another framework to query xml!". In *Demo Session Proceedings of SIGMOD'01*, 2001.

[12] W3C. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases, 2002.

[13] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics, 2002.

[14] Xin Zhang. XML Algebra for Rainbow System. Technical report, Worcester Polytechnic Institute, 2002. to appear.

[15] X. Zhang, G. Mitchell, W.-C. Lee, and E. A. Rundensteiner. Clock: Synchronizing internal relational storage with external xml documents. In K. Aberer and L. Liu, editors, *Eleventh International Workshop on Research Issues in Data Engineering: Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, 1-2 April 2001*, pages 111–118. IEEE Computer Society, 2001.

[16] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and B. Pielech. Rainbow: Mapping-Driven XQuery Processing System. In *Demo Session Proceedings of SIGMOD'02*, 2002.

A XAT Operators

XML operators (as shown in Table 3) are used to represent the XML document related operations, e.g., navigation and construction. SQL operators (as shown in Table 5) correspond to the relational complete portion of the algebra. Special operators (as shown in Table 4) include operators used temporarily in different phases of optimization and operators shared by the class of XML and of SQL operators. Each table describes the operator's name, its symbol, its parameters, the notion of the output columns, and its sources of data and subqueries, with their description. We show only the related operators in this paper, for the complete list of the XAT operators, please refer to [14].

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Expose	ϵ	*col*	N/A	s	Expose column col as XML documents or fragments.
Tagger	T	p	*col*	s	Taggering s according to list pattern p.
Navigate	ϕ/Φ	$col1, path$	$col2$	s	Navigate into column col of s based on path $path$.
Aggregate	Agg	*col*	N/A	s	Make a collection for each column. col is the focus column name.
XML Union	$\overset{x}{\cup}$	$col+$	*col*	s	Union multiple columns into one.
XML Intersect	$\overset{x}{\cap}$	$col+$	*col*	s	Intersect multiple columns into one.
XML Difference	$\overset{x}{-}$	$col+$	*col*	s	Compute the difference between two columns.

Table 3: XML XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
SQLstmt	SQL	$stmt$	$col+$	N/A	Execute a SQL query statement $stmt$ to underlying database.
Function	$\{F\}$	$param+$	*col*	$s?$	XML or user defined function over optional input table with given parameters.
Source	S	$desc$	$col+$	N/A	Identify a data source by description $desc$. It could be a piece of XML fragment, an XML document, or a relational table.
Name	$\rho, \rightarrow$	$col1, col2$	N/A	s	Rename column $col1$ of source s into $col2$.
Name	$\rho, \rightarrow$	ns	N/A	s	Rename table s into new name ns.
FOR	FOR	*col*	N/A	s, sq	FOR operator iterate over s and execute subquery sq with the variable binding column.
IF_THEN_ELSE	IF	c	N/A	sq_1, sq_2	If condition c is true, then execute subquery sq_1, else execute subquery sq_2.
Merge	M	N/A	N/A	$s+$	Merge multiple tables into one table based on tuple order.

Table 4: Special XAT Operators.

Operator	*Sym*	*Prms.*	*Output*	*Data*	*Description*
Project	π	$col+$	N/A	s	Project out multiple columns from input table s.
Select	σ	c	N/A	s	filter input table s by condition c.
Cartesian Product	$\times$	N/A	N/A	s_1, s_2	Cartesian product of the results of two input tables, s_1 and s_2.
Theta Join	$\bowtie$	c	$col+$	ls, rs	Join two input tables ls and rs under condition c, and output selected columns.
Outer Join	$\overset{\circ}{\bowtie}_L, \overset{\circ}{\bowtie}_R$	c	$col+$	ls, rs	Left (right) outer join two input tables ls and rs, and output selected columns.
Distinct	δ	$col+$	N/A	s	Eliminates the duplicates in the columns $col+$ of input table s by grouping.
Groupby	γ	$col+$	N/A	s, sq_g	Making temporary groups by multiple columns from input table s, then evaluate subquery sq_g for each group, then merge the evaluated results back.
Orderby	τ	$col+$	N/A	s	Sort input table s by multiple columns.
Union	$\cup$	N/A	N/A	$s+$	Union multiple sources together.
Outer Union	$\overset{\circ}{\cup}$	N/A	N/A	$s+$	Outer union multiple sources together.
Difference	$-$	N/A	N/A	ls, rs	Difference between two sources.
Intersect	$\cap$	N/A	N/A	$s+$	Intersect multiple sources.

Table 5: SQL XAT Operators.